Portable stateful big data processing in Apache Beam

Kenneth Knowles

Apache Beam PMC
Software Engineer @ Google
klk@google.com / @KennKnowles

https://s.apache.org/ffsf-2017-beam-state

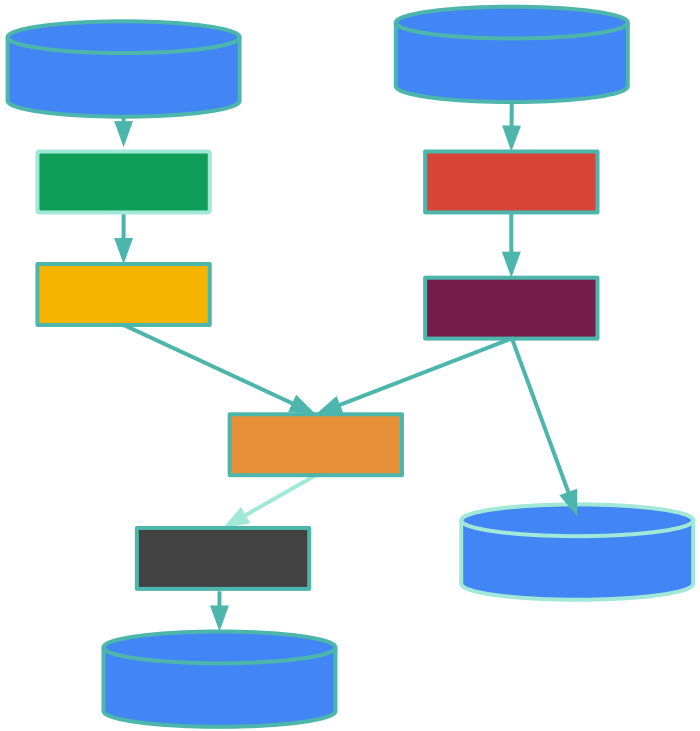
Flink Forward San Francisco 2017

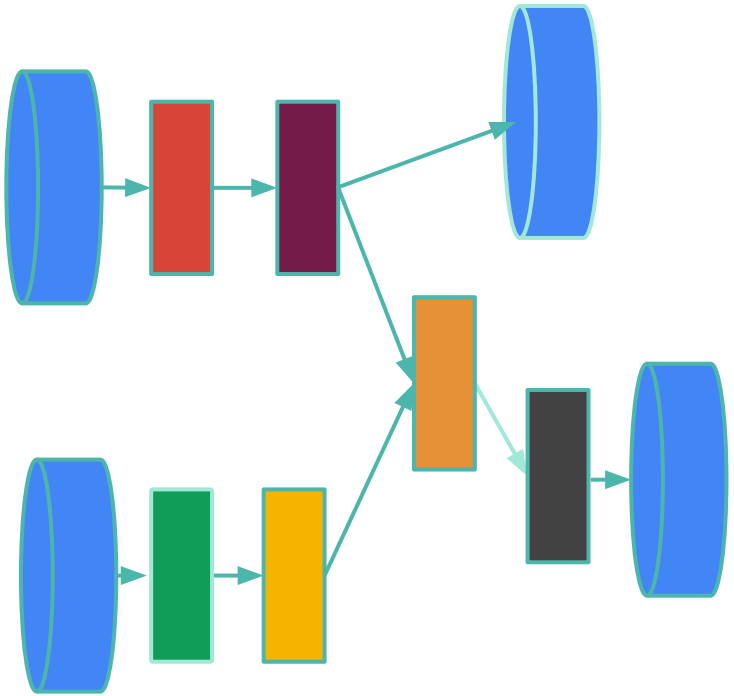
Agenda

1. What is Apache Beam?
2. State
3. Timers
4. Example & Little Demo

What is Apache Beam?

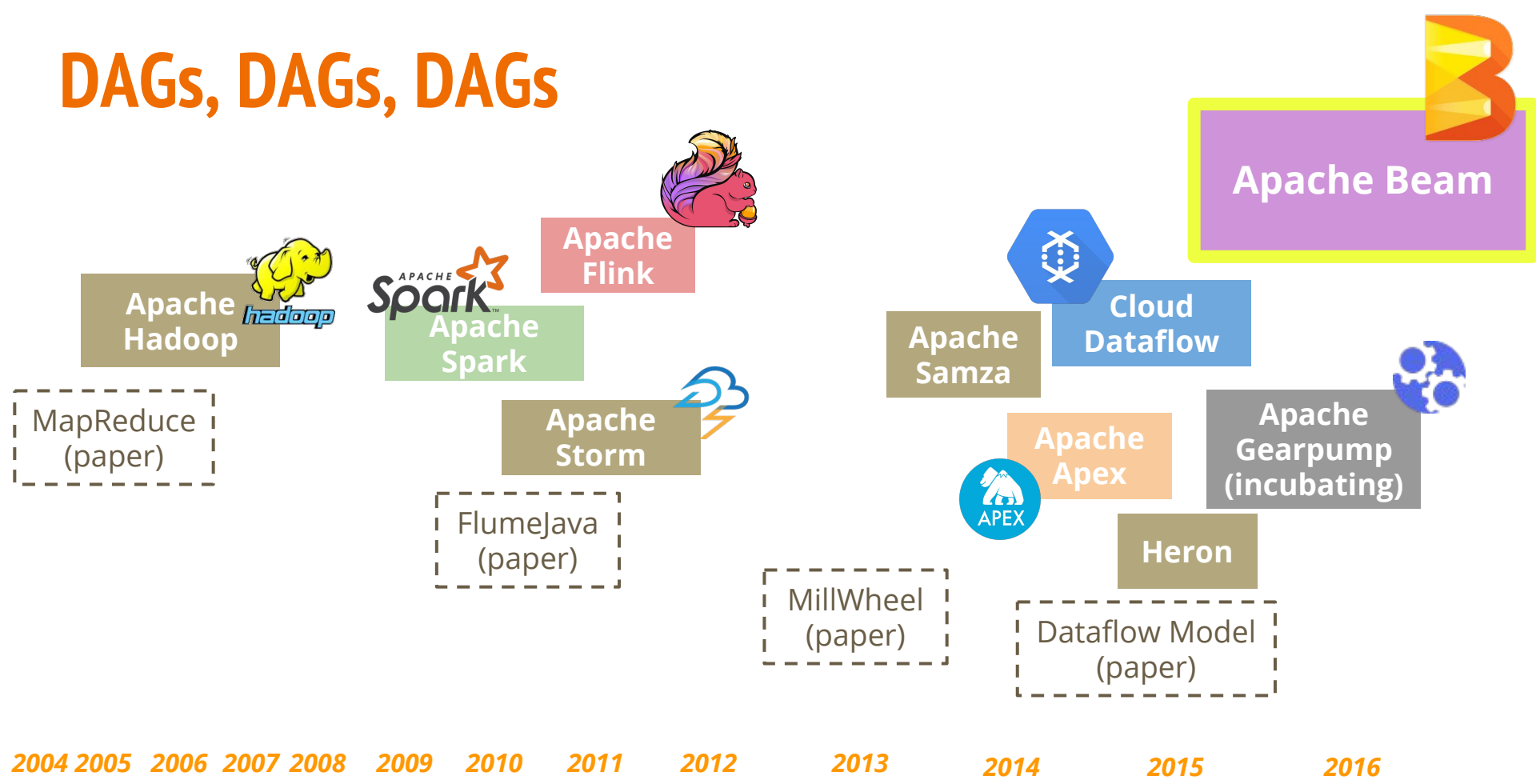
TL;DR





(Flink draws it more like this)

DAGs, DAGs, DAGs



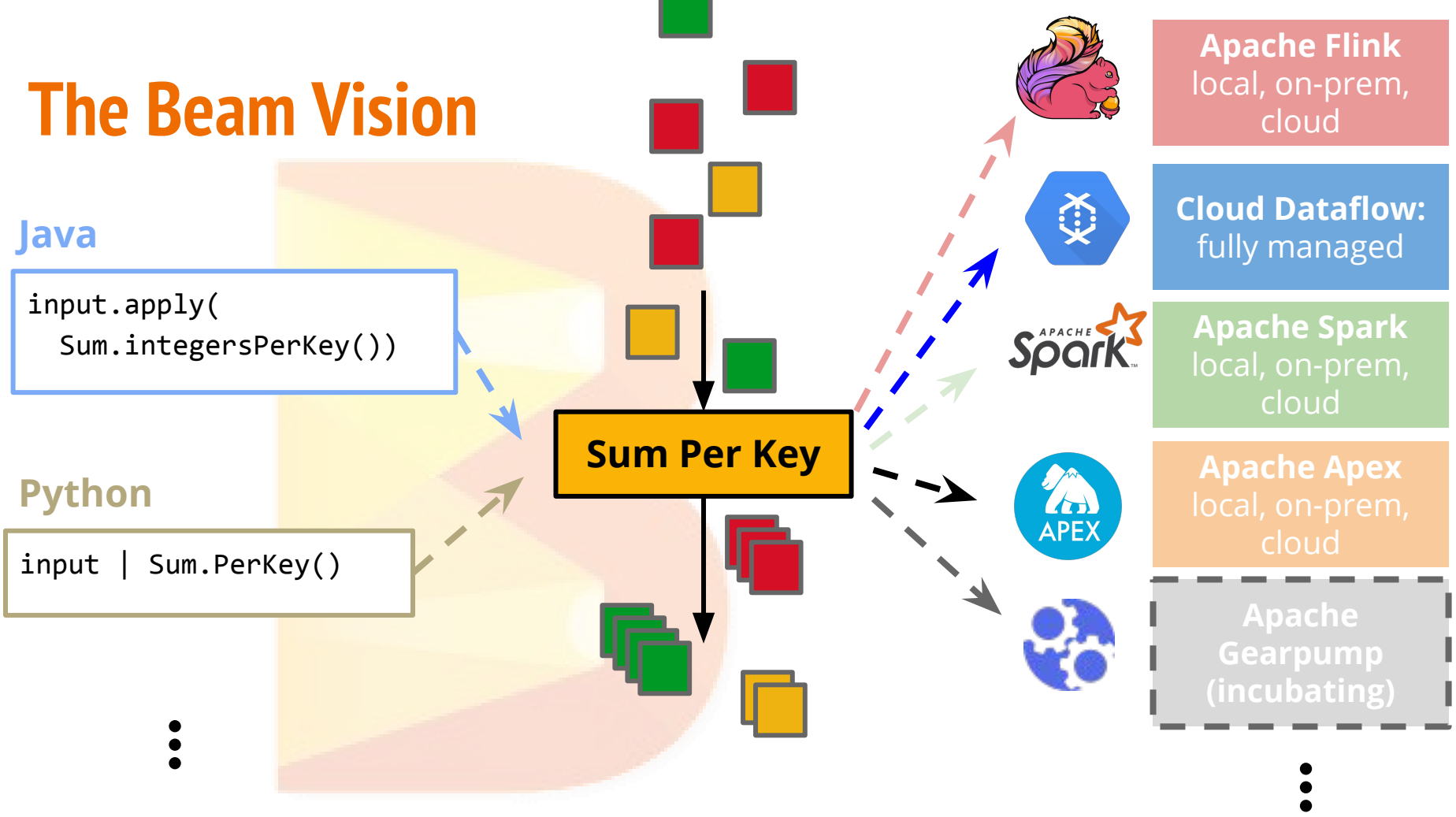
The Beam Vision

Java

```
input.apply(
  Sum.integersPerKey())
```

Python

```
input | Sum.PerKey()
```



Apache Flink
local, on-prem, cloud

Cloud Dataflow:
fully managed

Apache Spark
local, on-prem, cloud

Apache Apex
local, on-prem, cloud

Apache Gearpump (incubating)

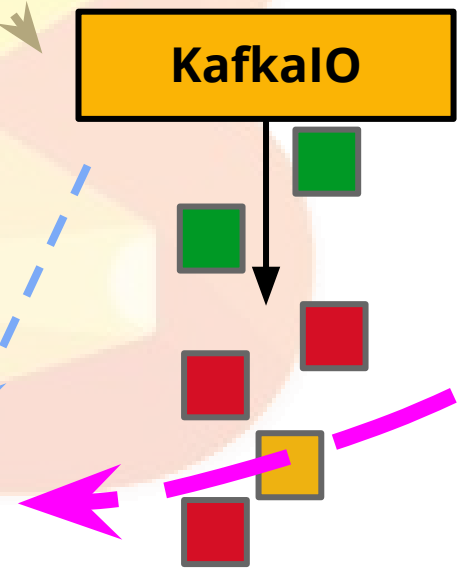
The Beam Vision

Python

```
input | KakaIO.read()
```

Java

```
class KafkaIO extends
UnboundedSource { … }
```



Apache Flink
local, on-prem, cloud

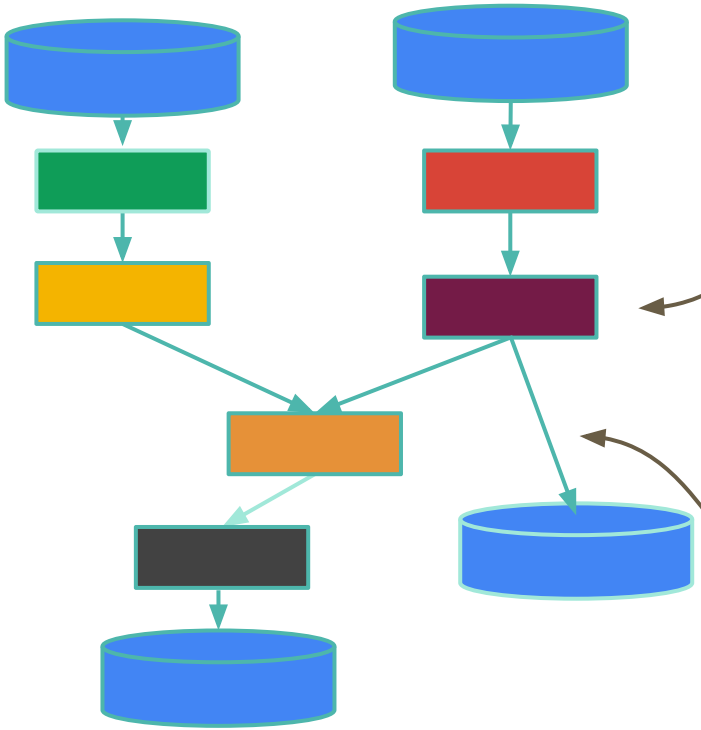
Cloud Dataflow:
fully managed

Apache Spark
local, on-prem, cloud

Apache Apex
local, on-prem, cloud

Apache Gearpump (incubating)

The Beam Model



PTransform

Pipeline

PCollection
(bounded or unbounded)

The Beam Model

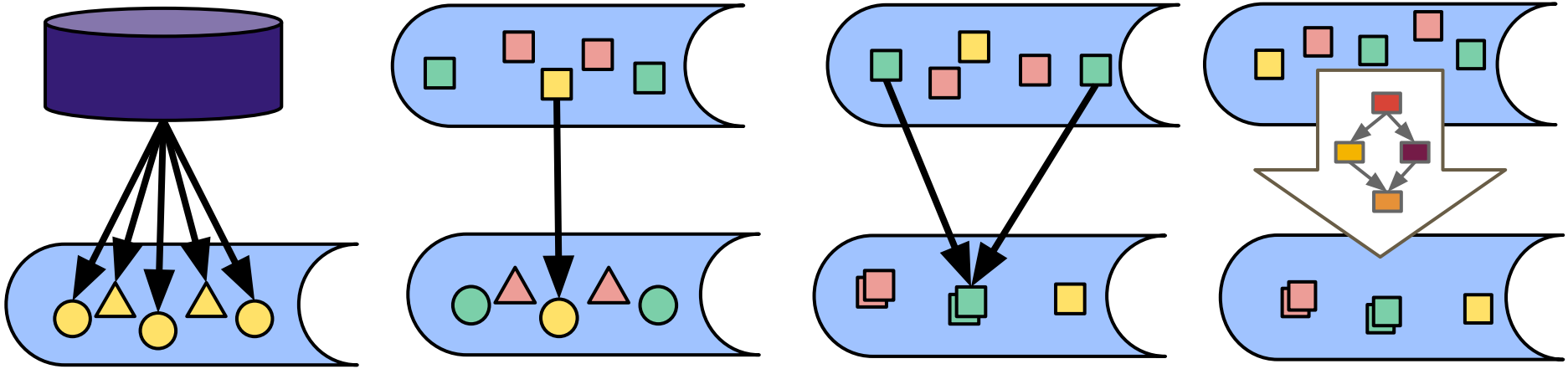
What are you computing? (read, map, reduce)

Where in event time? (event time windowing)

When in processing time are results produced? (triggers)

How do refinements relate? (accumulation mode)

What are you computing?



Read
Parallel connectors to external systems

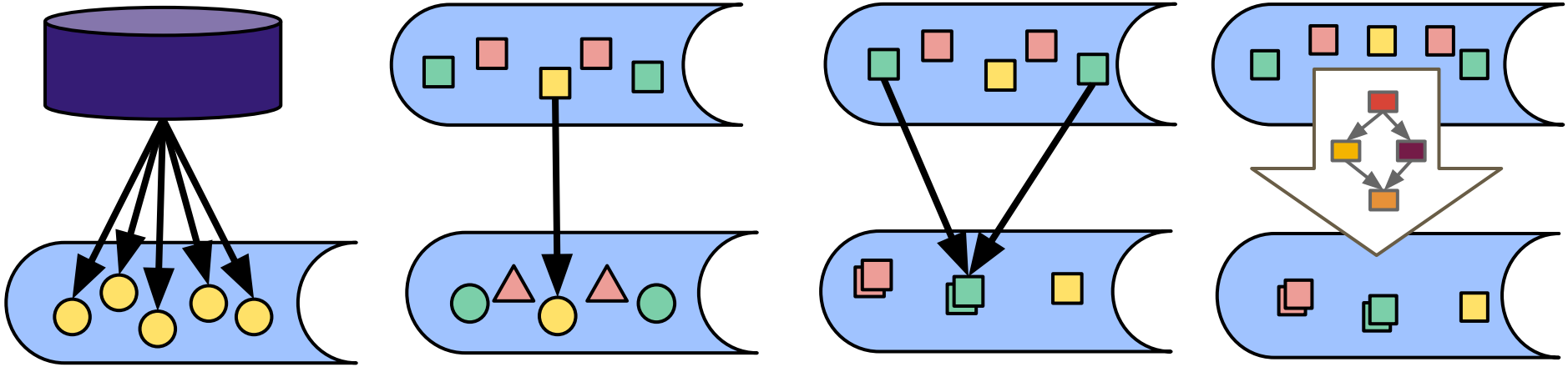
ParDo
Per element "Map"

Grouping
Group by key, Combine per key, "Reduce"

Composite
Encapsulated subgraph

State

What are you computing?



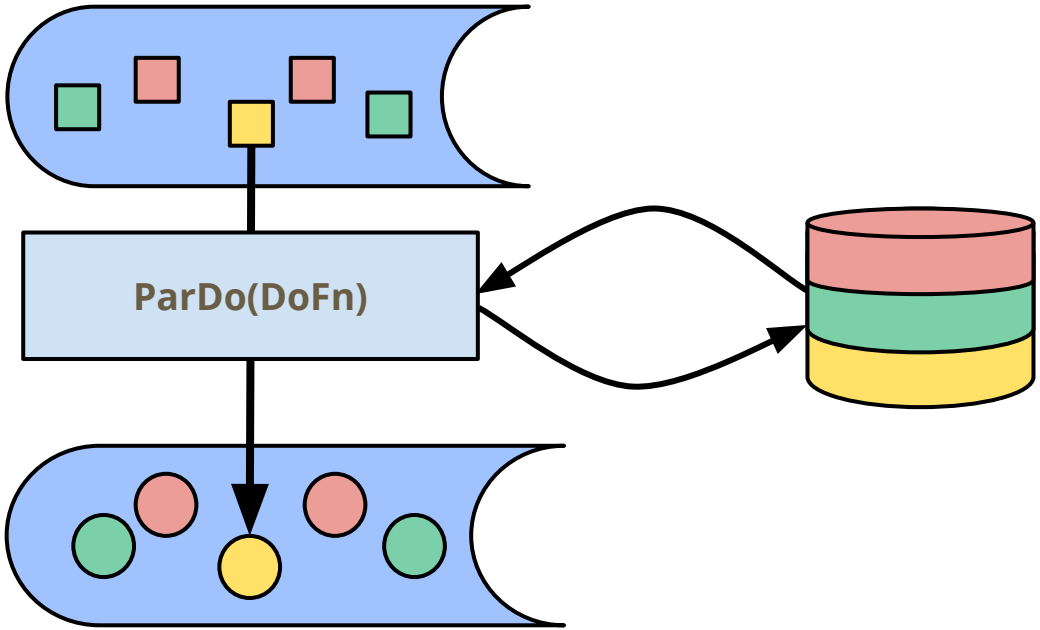
Read
Parallel connectors to external systems

ParDo
Per element "Map"

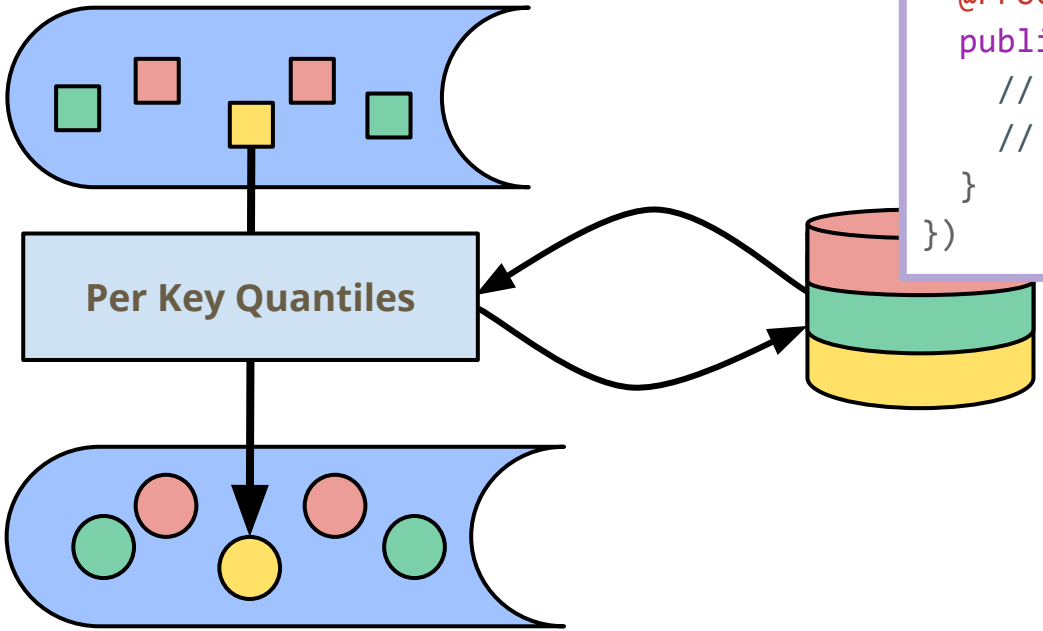
Grouping
Group by key, Combine per key, "Reduce"

Composite
Encapsulated subgraph

State for ParDo

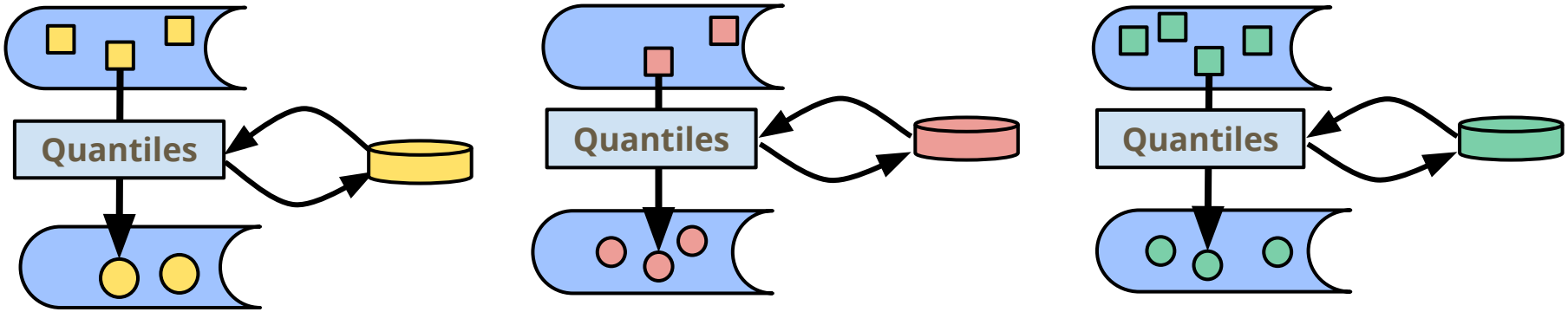


"Example"



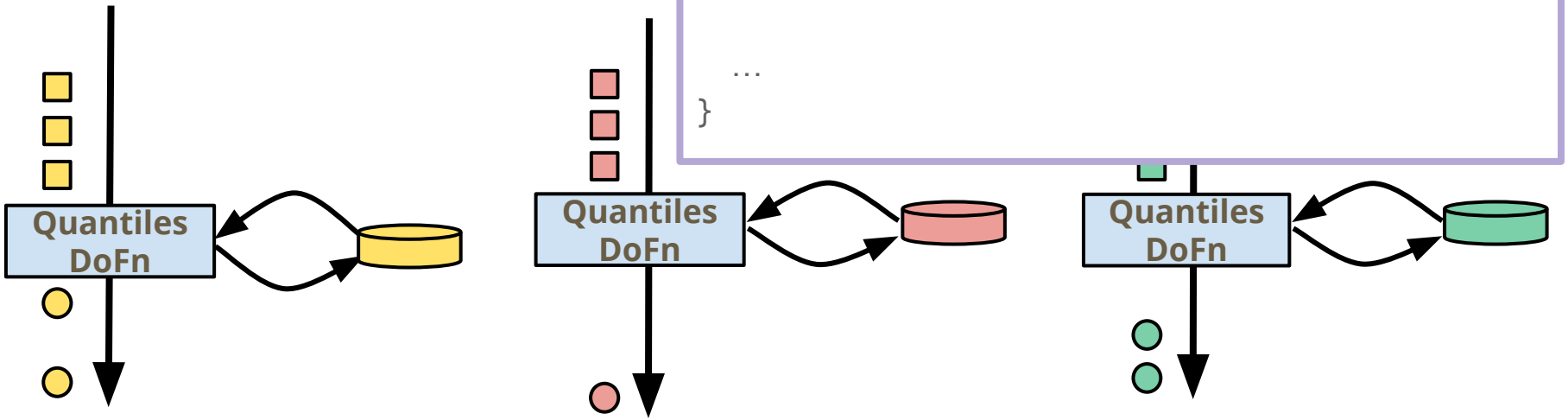

```
ParDo.of(new DoFn<...>() {
  // declare some state

  @ProcessElement
  public void process(...) {
    // update quantiles
    // output if needed
  }
})
```

Partitioned

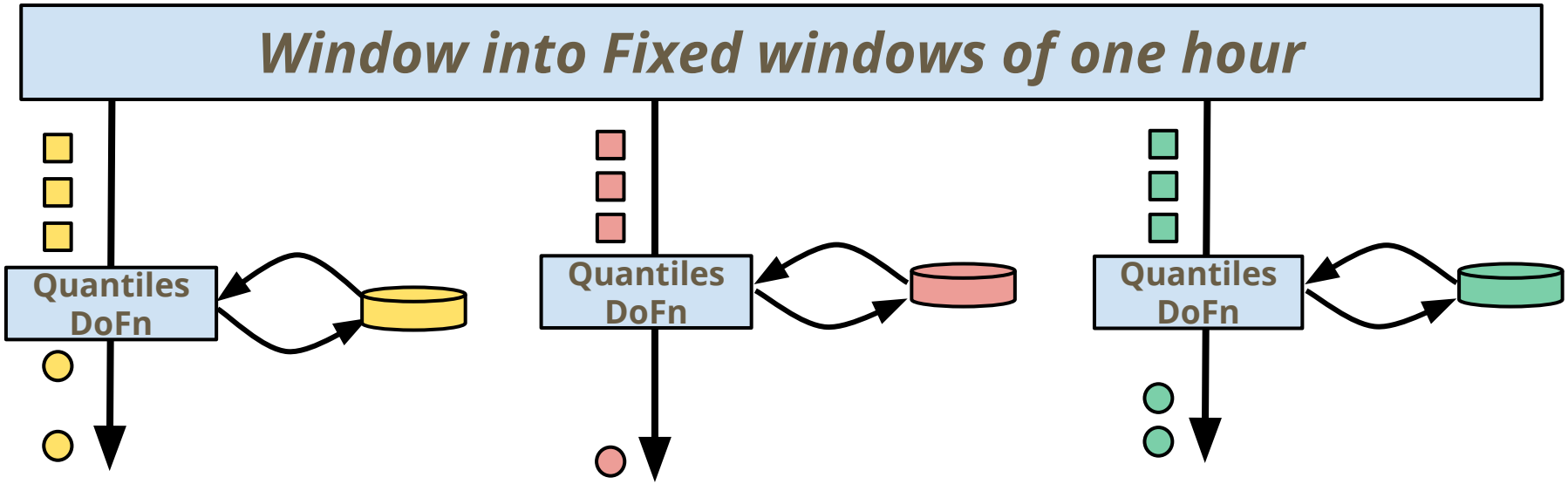


Parallelism!

```
new DoFn<Foo, Quantiles<Foo>>() {
  @StateId("quantiles")
  private final StateSpec<...>
      quantilesState = StateSpecs.combining();

  ...
}
```

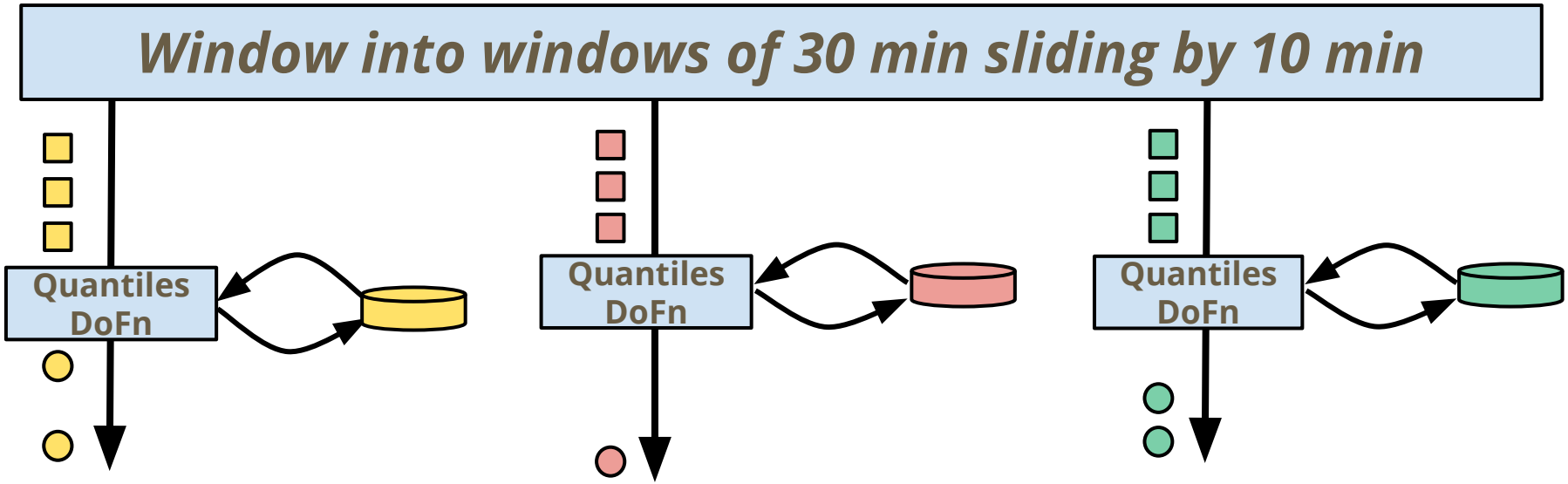


Windowed



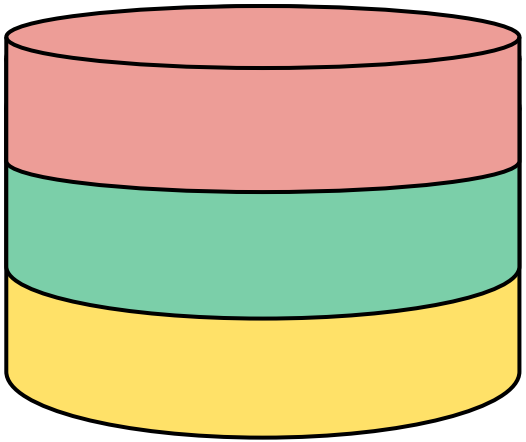
Expected result: Quantiles for each hour

Windowed



Expected result: Quantiles for 30 minutes sliding by 10 min

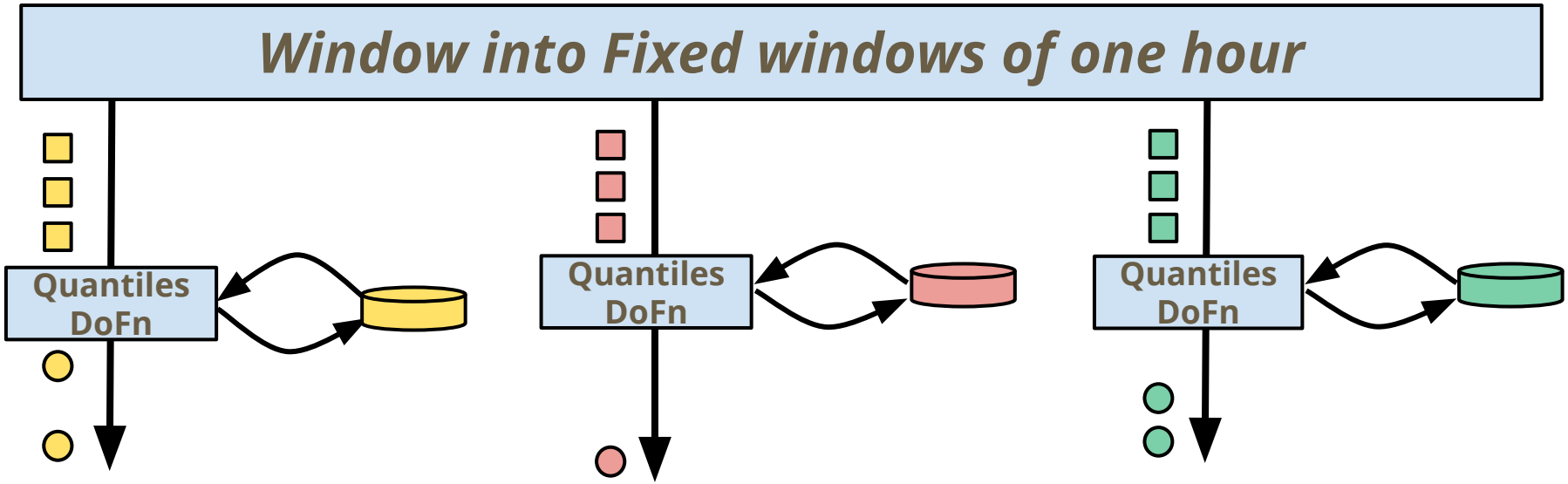
State is per key and window



	<k, w>$_1$	<k, w>$_2$	<k, w>$_3$	...
"x"	3	7	15	
"y"	"fizz"	"7"	"fizzbuzz"	
...				

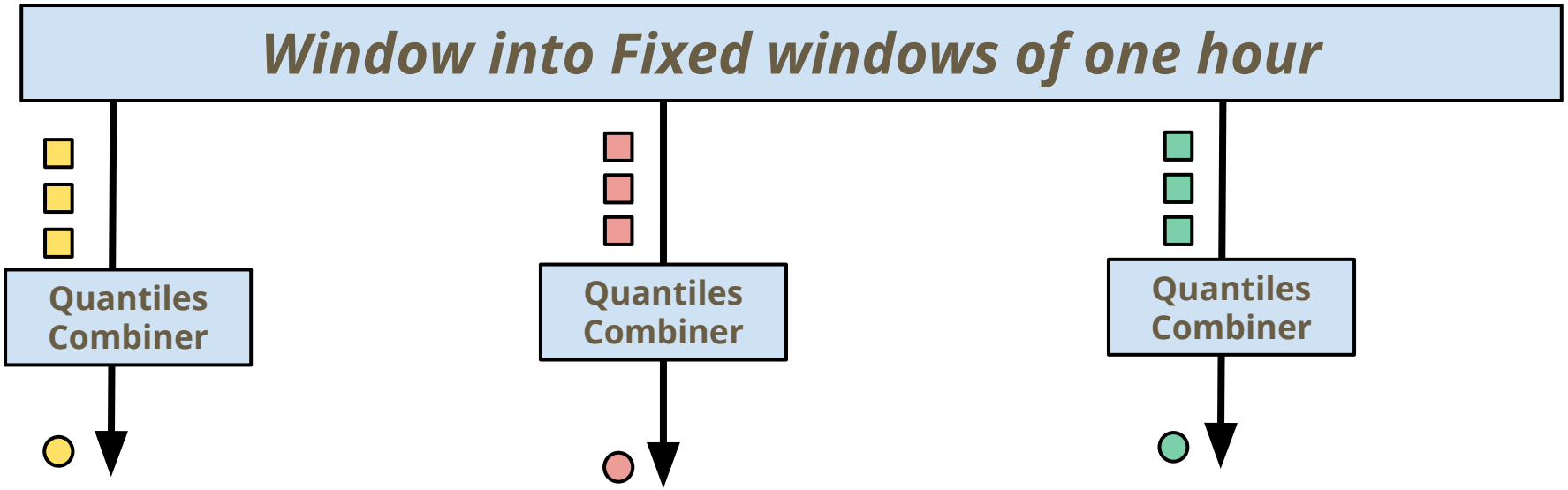
Bonus: automatically garbage collected when a window expires
(vs manual clearing of keyed state)

Windowed



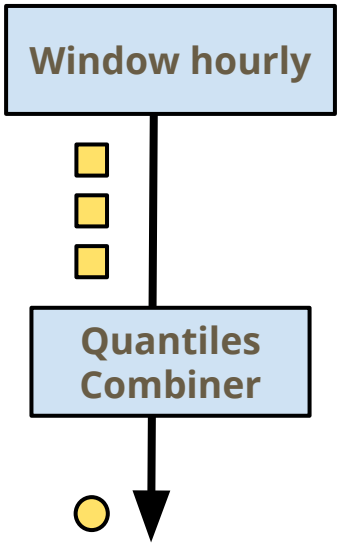
Expected result: Quantiles for each hour

What about Combine?

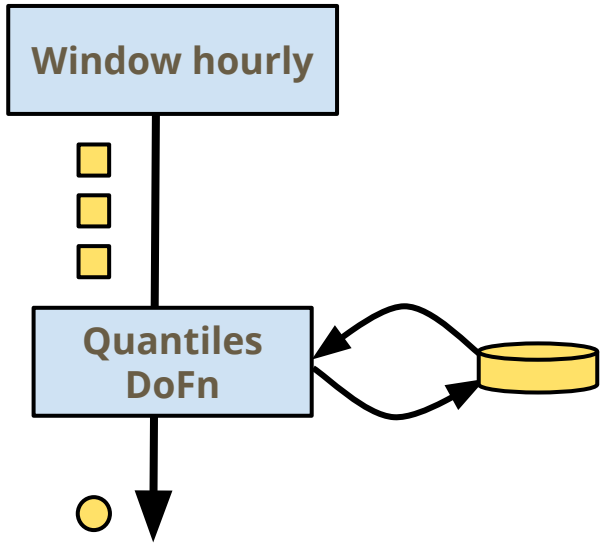


Expected result: Quantiles for each hour

Combine vs State (naive conceptualization)

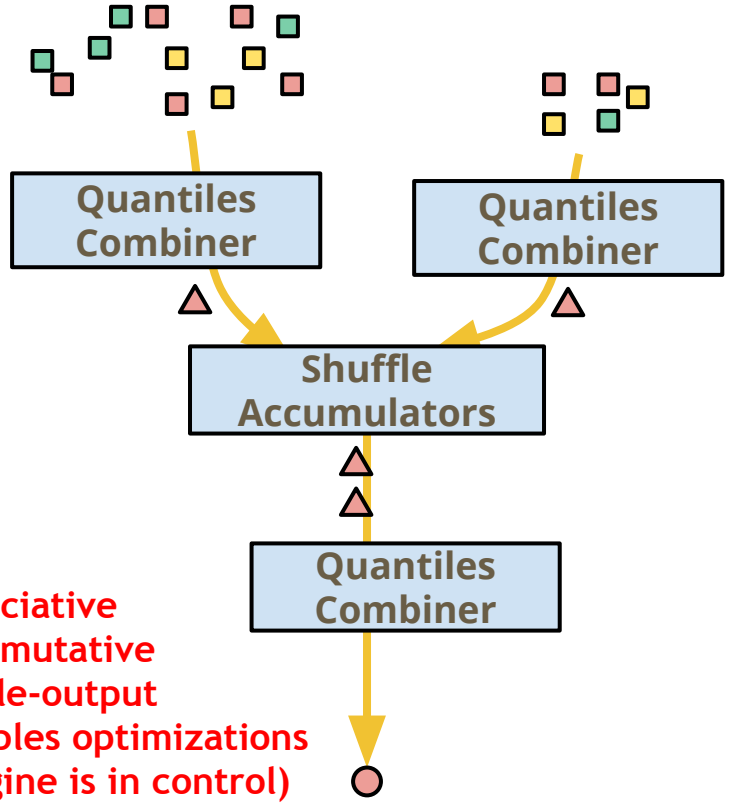


Expected result:
Quantiles for each hour

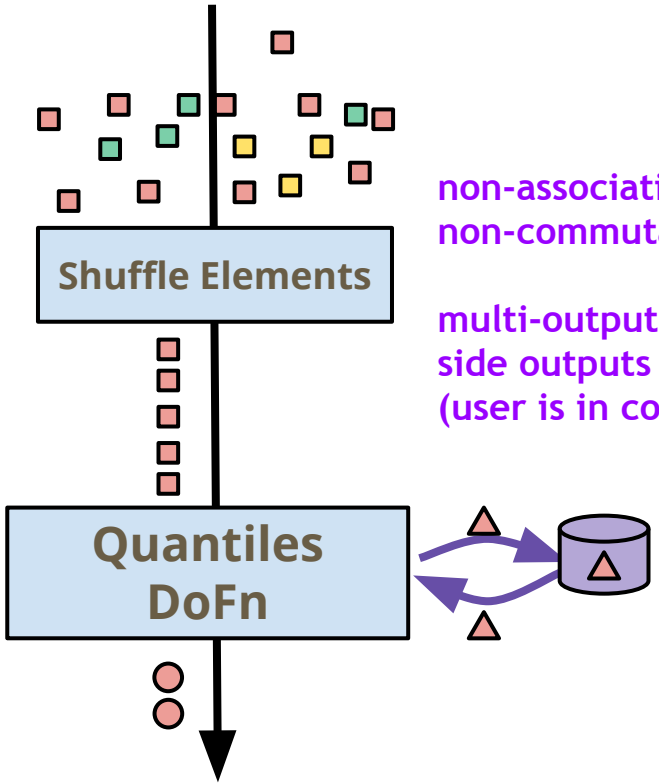


Expected result:
Quantiles for each hour

Combine vs State (likely execution plan)



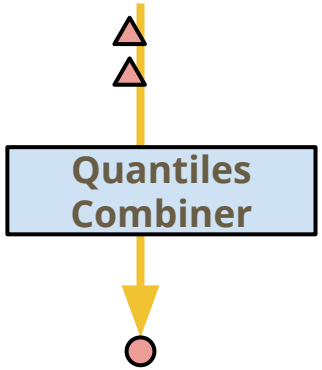
associative
commutative
single-output
enables optimizations
(engine is in control)



non-associative*
non-commutative*

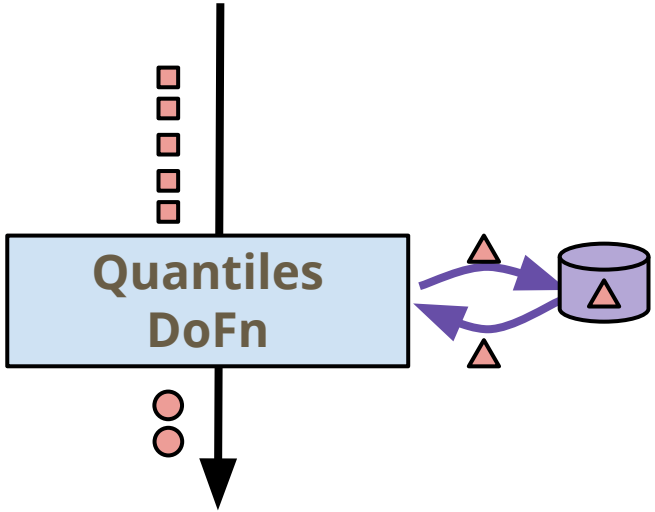
multi-output
side outputs
(user is in control)

Combine vs State



output governed by trigger

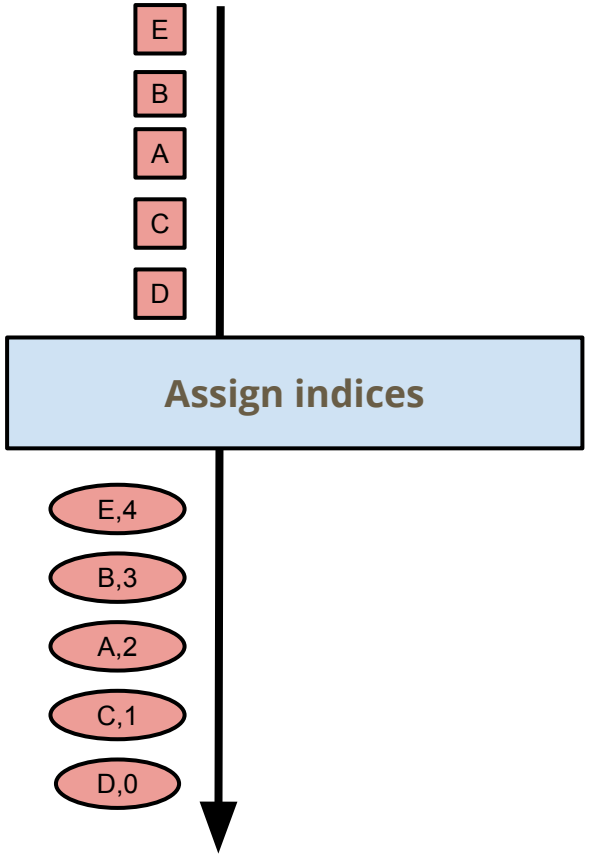
(data/computation unaware)



"output only when there's an interesting change"

(data/computation aware)

Example: Arbitrary indices



non-associative

non-commutative

non-deterministic

and totally fine!

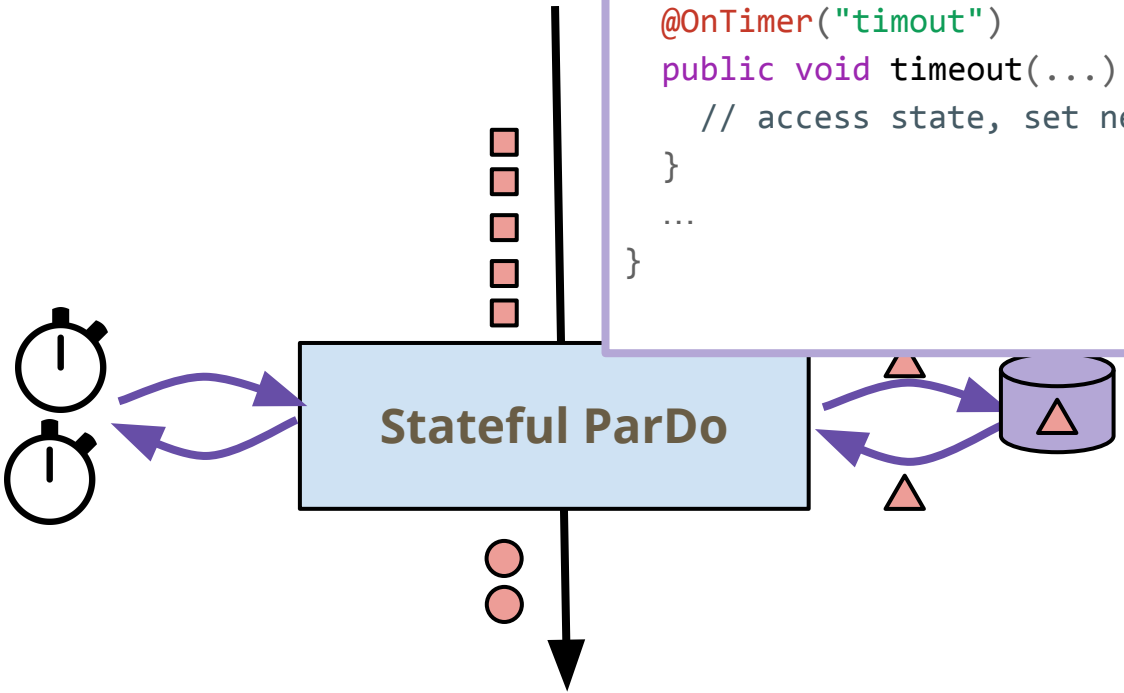
Kinds of state

- `Value` - just a mutable cell for a value
- `Bag` - supports "blind writes"
- `Combining` - has a CombineFn built in; can support blind writes and lazy accumulation
- `Set` - membership checking
- `Map` - lookups and partial writes

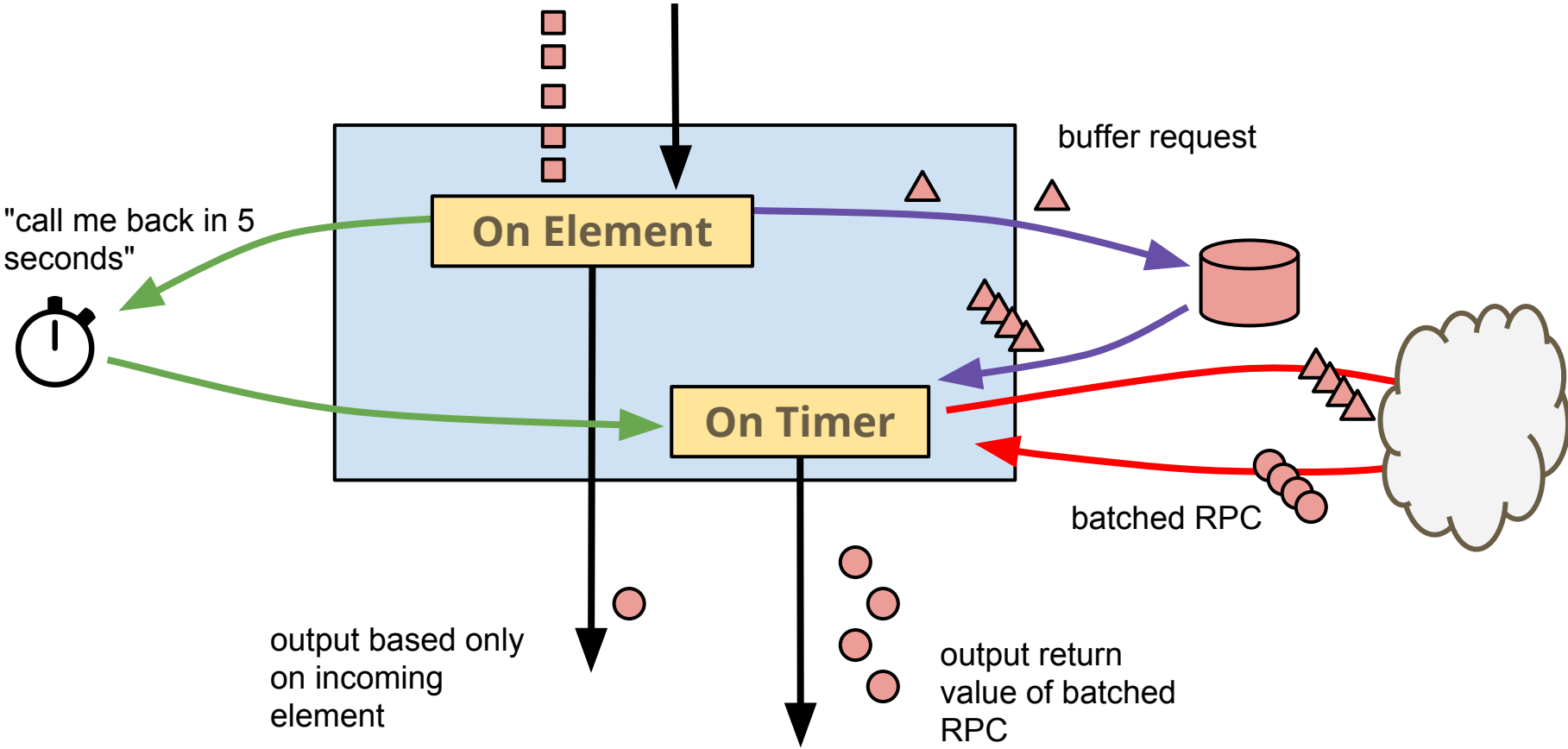
Timers

Timers for ParDo

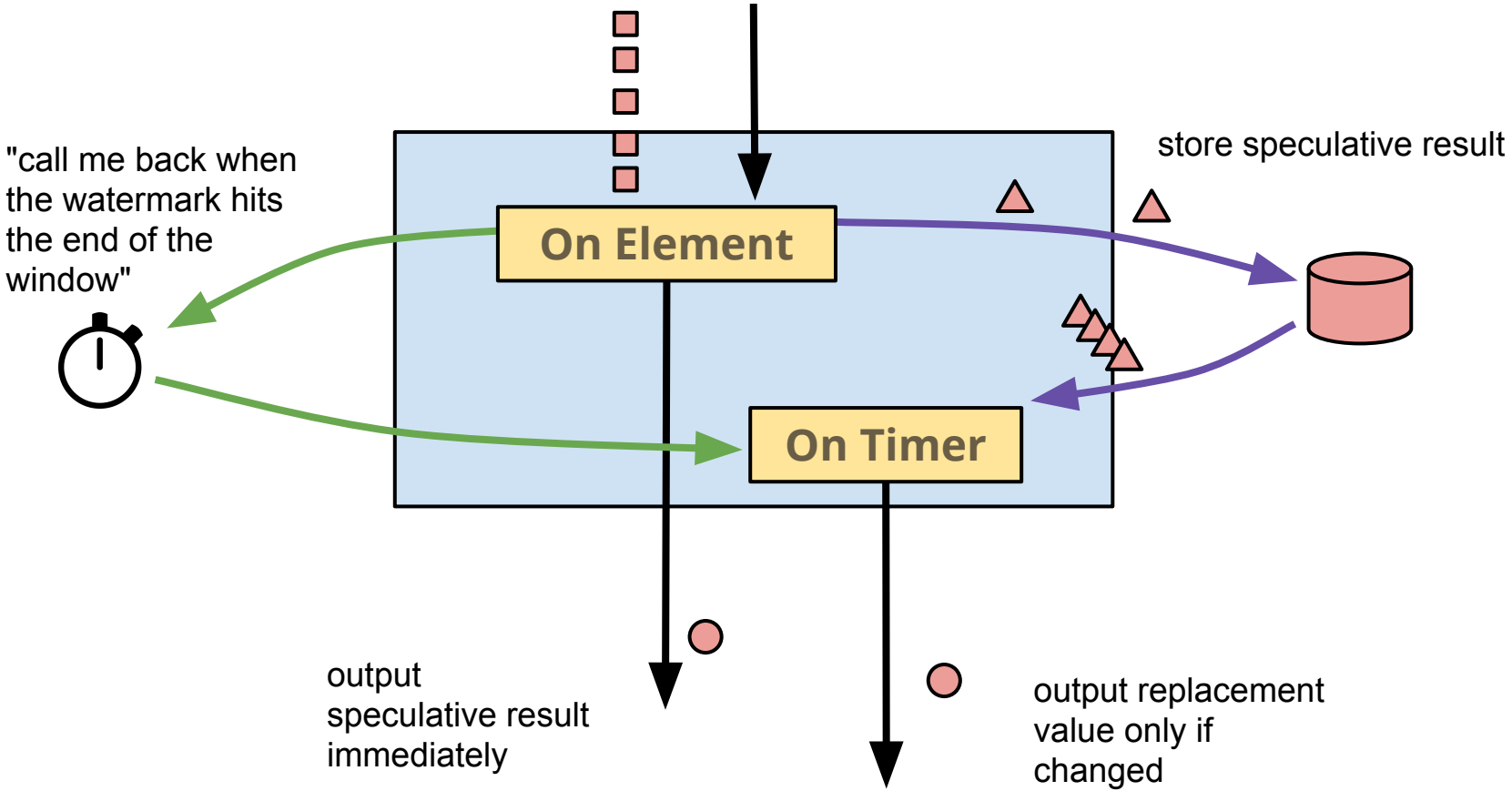
```
new DoFn<...>() {
  @TimerId("timeout")
  private final TimerSpec timeoutTimer =
    TimerSpecs.timer(TimeDomain.PROCESSING_TIME);

  @OnTimer("timout")
  public void timeout(...) {
    // access state, set new timers, output
  }
  …
}
```



Timers in processing time



Timers in event time



More example uses for state & timers

- Per-key arbitrary numbering
- Output only when result changes
- Tighter "side input" management for slowly changing dimension
- Streaming join-matrix / join-biclique
- Fine-grained combine aggregation and output control
- Per-key "workflows" like user sign up flow w/ expiration
- Low-latency deduplication (let the first through, squash the rest)

Performance considerations (cross-runner)

- Shuffle to colocate keys
- Linear processing of elements for key+window
- Window merging
- Storage of state and timers
- GC of state

Demo

Summary

State and Timers in Beam...

- ... unlock new uses cases
- ... they "just work" with event time windowing
- ... are portable across runners (implementation ongoing)

Thank you for listening!

This talk:

- Me - @KennKnowles
- These Slides - https://s.apache.org/ffsf-2017-beam-state

Go Deeper

- Design doc - https://s.apache.org/beam-state
- Blog post - https://beam.apache.org/blog/2017/02/13/stateful-processing.html

Join the Beam community:

- User discussions **-** user@beam.apache.org
- Development discussions - dev@beam.apache.org
- Follow **@ApacheBeam** on Twitter

You ***can contribute to Beam + Flink***

- New types of state
- Easy launch of Beam job on Flink-on-YARN
- Integration tests at scale
- Fit and finish: polish, polish, polish!
- ... and lots more!

https://beam.apache.org